Redeye

Documentation for Redeye (ComLynx) support code
Date: 15-Oct-90

Corrections and Additions

The previous release (March 1990) of the Redeye documentation incorrectly described the sign of the PlayerFlag0 and PlayerFlag1 bytes.

The constant REDEYE_SLOWNESS has been added to the Redeye system code, and is now described in this document (see the Resource Usage and Optimization sections).

The switch SPLIT_SEND_RECEIVE has been added to the example "glue" interface code, and is now described in this document (see the Optimization section).

A new list of the "currently in use" Redeye ID numbers is included.

Introduction

At last, it's here! The long awaited redeye (ComLynx) communication release.

Redeye does not come without some costs, however. Installation is not trivial, so I will try to describe what redeye does, doesn't do and how to install and use it.

Please be sure to read through this document before attempting to install redeye in your code.

*NOTE: While I have attempted to be accurate in this document, there may still be errors. The example program TESTREDEYE.SRC in the 6502:examples directory is known to work, and should be used as a reference if you are having difficulties.

Overview

Redeye communication is handled in a round-robin communication-frame based system. When all is going well, player 0 (the master) gives out his information, followed immediately by player 1, player 2, etc. The acknowledge of receipt of messages is implied by sending new data out. Even if a unit has nothing new to add to the state of the universe, he

ust send out a message so that everyone else knows their messages have ll been received. If there is any problem, the master acts as the director prompting units for their data if they are late sending, and retransmitting data if anyone else requests something they have missed. This is all handled by the system routines during interrupts, and does not need to be dealt with by the game program. To keep the process going smoothly, a handshake protocol must be obeyed between the game code and the system communication software. This handshake has been isolated to a few glue routines in redeye_glue.src in the 6502:examples directory.

To get to a normal communication state, the system needs information about other players, so the process starts with a logon. Redeye logon uses a system very similar to many ethernet style two-wire communication systems. Each unit initially assigns itself player number zero, waits for a while, and then announces its identity to the world. If a unit hears someone else claim the same number the unit thinks it has, it picks a new player number that it hasn't heard claimed. After any communication, a unit waits a random amount of time (short if it has the next sequential number, intermediate if it has the next used number but unused ones precede it, and long if someone else is to speak next). Randomness is thrown into the time delays so that if two units think they have the same number, they won't speak at identical times.

When all units agree about the state of the system, and a user has requested logon termination, the request is acknowledged and the process ends. If a user has requested termination, but units don't agree, termination will be delayed, and may be denied awaiting another termination request. After logon is complete, redeye shuts down completely allowing the game that needs the space to overwrite the logon specific code before starting up normal communication.

*NOTE: The term "game frame" or sometimes just "frame" is used in document. This is not to be confused with the 60Hz video frame, or the 240Hz audio frame in the Lynx. A "game frame" (usually) consists of one pass through the main loop of the game, including reading of inputs and generating a display. The game frame is frequently longer than one video frame and they may not be related to each other in any way.

TESTREDEYE.SRC Example

The testredeye.src program can be assembled and downloaded to see a very simple example of redeye usage.

The first screen that comes up is the logon screen. It displays a message in the center of the display ("LOGON IN PROGRESS") and two single hex digits below the message. The digit on the left shows the player number currently assigned to this unit, and the right digit shows 1 less than the total number of units logged in. Pressing A or B button on any unit should exit logon, and enter the "game".

On the "game" screen, a number of colored arrowhead shapes fly out rom the center in different directions. A set of two-digit hex numbers are displayed on the left edge of the screen (colored the same as the

lying shapes). Every time a player pushes down on their joystick or resses a button on their unit (except PAUSE), their number will ncrement. Pressing left or right on a joystick causes that player's shape to spin around.

FLIP and RESET are supported in the example program.

Installation and Use

To install redeye, you need to include the following files in your code:

```
6502:include/msgmgr.i

6502:macros/msgmgr.mac

6502:src/comlink.src
6502:src/comlink_logon.src
6502:src/msgmgr.src

6502:src/comlink_variables.src
6502:src/msgmgrRam.src
```

The file comlink_logon.src can be included within a segment that is overwritten after logon is complete. The other .src files must be included where they will be resident both during logon and normal communication. The files comlink_variables.src and msgmgrRam.src reserve space for variables and are best included either at the beginning or end of a segment so that space does not have to be reserved in the cartridge ROM. Also be aware that msgmgr.src includes some self-modifying code.

The redeye system files require that the user define the following constants and swithes before including:

SERIALPORT_USER - This switch should be defined to let the system know that you use the serial (redeye) port.

GAME_ID - The unique game ID number. Range: 1 to 65534
The following numbers are in use:

0	- test programs
1	- Gauntlet the Third Encounter
2	- Zarlor Mercenary
3	- California Games
4	- Xenophobe
5	- Slime World
6	- RoboSquash
7	- Warbirds (Space Wars?)
83	- Shanghai
4949	- Rampage
65535	- test programs

MAX_PLAYERS - The maximum number of players this game allows.
Range: 2 to 16

PLAYER_DATA_SIZE - The maximum size of data contained in a player's message. Range: 1 to 256/MAX_PLAYERS

VAR_SIZE_DATA - This switch enables code to deal with variable sized messages. Only define if you want variable size messages.

REDEYE_SLOWNESS - This constant is used to decrease the speed of Redeye communication to survive better in a system with very heavy IRQ usage (digitized sounds, for example). Set to 0, 1, 2 or 3 to select 62500, 31250, 15625, or 7812.5 baud.
Range: 0 to 3

The redeye routines also require that you provide (but not write to) the following variables (preferably in zero page):

NumberOfPlayers - number of players logged in -1
PlayerNumber - player number assigned to this unit

* * * * * * * *

NOTE: At this point, if you only want to pass around joystick and button information in your program, you might want to skip down to the "Quick installation" subsection.

* * * * * * * *

To start the logon process, call the system routine start_logon . The logon process will run unattended from the interrupt routines. The game should give some feedback to the user about what is going on. The following variables are used for communication to/from the logon code:

EndLogonRequest - request to end logon, game code sets to non-zero to request logon termination
LogonInProgress - status of logon, non-zero during logon reset when logon is over
NumberOfPlayers - 1 less than number of players logged in
PlayerNumber - current number assigned to this unit

The game code should only write to EndLogonRequest . The logon process may reject a request for termination, and the request will need to be re-asserted. PlayerNumber may change during logon, and may even be set to a higher number than NumberOfPlayers. During logon termination, player numbers will be packed down, and after logon PlayerNumber will be no larger that NumberOfPlayers. Logon may terminate from any machine's request, so the game code should poll LogonInProgress even if EndLogonRequest has not been set.

The routine checklogonover in redeye_glue.src shows an example of interface to the logon code, where termination is requested by players hitting the A or B buttons. The loop in the middle of the routine showlogonscreen in testredeye.src shows a simple example of a display that conveys some information about the logon process.

After logon, when the game is ready to begin, call the system routine start_comlink to re-install the redeye interrupts and prepare for normal communication. During normal communication, the following variables are used for communication and handshake to and from the system redeye routines:

OutGoingData - buffer for outgoing data

Seq - flag for whether on even/odd data sequence,
zero means read ODD sequence data,
non-zero means read EVEN sequence data

PlayerData0 - incoming even sequence data, arranged as
MAX_PLAYERS number of sequential arrays with
PLAYER_DATA_SIZE number of elements in each
PlayerData1 - incoming odd sequence data, arranged as
MAX_PLAYERS number of sequential arrays with
PLAYER_DATA_SIZE number of elements in each

PlayerFlag0 - array of flags for whether players' even
sequence data has come in/has been used
positive means valid data to read,
game code sets to negative to acknowledge data
used
PlayerFlag1 - array of flags for whether players' odd
sequence data has come in/has been used
positive means valid data to read,
game code sets to negative to acknowledge data
used

The following variables are used when the VAR_SIZE_DATA switch is set:

OutGoingSize - how many bytes of data in OutGoingData
range: 0 to PLAYER_DATA_SIZE

PlayerDataSize0 - array for each player of how many bytes
received in PlayerData0
PlayerDataSize1 - array for each player of how many bytes
received in PlayerData1

In addition, the constant array PlrOffsets is an array of offsets into the PlayerData0 and PlayerData1 arrays to the beginning of a player's data.

For each frame of communication, the game code should put its outgoing data in the array OutGoingData and set OutGoingSize (if using VAR_SIZE_DATA). Call the routine launch_redeye to signal to the system code that your data is ready. The routine preparejoysticks in redeye_glue.src shows an example of preparing data to send. The first frame of communication will require setting up outgoing data before processing incoming data, so in the example program do_start_comlink branches to preparejoysticks.

Incoming data is more complicated to deal with. Incoming data is accumulated in the PlayerData0 and PlayerData1 arrays for even and odd sequence frames. The variable Seq signals whether the game code should

deal with the even or odd sequence frames - note that Seq is zero when reading an ODD frame, and non-zero when reading an EVEN frame. The game code should deal with each player's data sequentially from player zero to the highest numbered player. The bytes in the arrays PlayerFlag0 and PlayerFlag1 are set positive when the appropriate frame data has been received. After the game code is through with the data from a player, it should set the PlayerFlag0 or PlayerFlag1 value to negative to acknowledge receipt and use of the data. After the flag is reset, the redeye code may immediately overwrite the data, so it can only be reset after the data is completely used. The PlayerData0 and PlayerData1 buffers are read-only and should not be modified by the game code. The array PlayerDataSize contains the size of the incoming messages (if VAR_SIZE_DATA is set). The routine getinput in redeye_glue.src shows an example of how to read and acknowledge data.

Quick Installation

For those games that only want to pass around joystick and button information, the file redeye_glue.src in the 6502:examples directory contains all of the interface code necessary to communicate with redeye. The example program testredeye.src in 6502:examples includes the sample glue code.

The provided glue code expects PLAYER_DATA_SIZE to be set to 2. Defining the switch VAR_SIZE_DATA will slightly increase code size, but will improve system performance. The glue code also needs the following variables and arrays to be provided:

restartflag	- 1 byte
restartcount	- 1 byte
joystick	- MAX_PLAYERS bytes
oldjoystick	- MAX_PLAYERS bytes
switch	- MAX_PLAYERS bytes
oldswitch	- MAX_PLAYERS bytes

In addition, the glue code calls the routine showlogonscreen which is assumed to be code which will display a screen giving some feedback to the user about the state of logon, and which should also handle double-buffering of the display. The variables NumberOfPlayers and PlayerNumber are available for feedback. See the example routine showlogonscreen in testredeye.src .

To start the logon process, call the routine do_logon . Control will be returned when logon is ended (by some player pressing the A or B button). Call the routine do_start_comlink sometime after logon, but before your main game loop begins. Call do_comlink_joysticks once per game frame to get communication about other players' inputs. RESET and FLIP are both handled by the glue code during logon and during normal communication. RESET assumes that the address restart will be provided as a location to jump to to restart the code.

After logon, NumberOfPlayers is set to one less than the number of players connected and PlayerNumber is set to the number assigned to the local unit. During gameplay, the arrays joystick and switch reflect the

urrent input values being used by each of the players. The arrays ldjoystick and oldswitch contain the previous game frame's values for inputs. The routine do_start_comlink initializes the elements of each of the arrays to $FF, and the routine do_comlink_joysticks copies the current values to the old, and gets new current values from the redeye communication buffers.

The glue code provided is just an example, but can be used directly by your code in its current state. Feel free to modify it, but make sure that you copy it out of the 6502: directory first.

Reset and Power-down Timeout

Game reset and redeye bring up some interesting issues. For the sake of reliability and stability of the communication routines, it is best if units reset together. Redeye will screw up if two games in progress with two or more players are connected together, and this situation is easy to achieve if units can reset locally and then start a new game, while other units have not reset. In general if the circumstances that signal reset are noticed, then one more frame of communication should be allowed before resetting so that all units have a chance to notice the same conditions. The program testredeye.src checks for players pressing reset in the game loop processplayers, and if found, sets a bit in restartflag. After the next time the glue routine getinput is called (allowing a complete communication frame), the flag is checked to see if the game should reset. Another situation arises if one player has disconnected or lost power. Communication will go down, and if everyone has to wait for a frame of communication before reset can be handled, they cannot reset. In testredeye the problem is solved by checking for local reset when waiting for incoming data, and if a local reset occurs, starting a timeout until deciding that communication is down and the system should be reset locally.

Lynx games should turn off after a period of inactivity. This is to both save batteries and to lengthen the lifetime of the system. The switch AUTO_TIMEOUT_USER enables some system code in the end of frame IRQ handler that checks for activity on the joystick, and if there is none for a long enough period of time, shuts off power to the system. Redeye games, however, have the potential for situations where one player is inactive while others are continuing on (he's completed a level early, died, or is just waiting for other players to catch up) and it would be very frustrating if his unit spontaneously shut off killing everyone's game. The example program solves this with the routine checkinactivity , which checks for activity on any unit and uses the system macro RESET_TIMEOUT to reset the timeout count. Games that don't pass joystick information may need more sophisticated checks.

Redeye Limitations

The Lynx hardware can electrically support up to 18 units connected together. For a number of reasons, 16 has been chosen for the maximum

for the number of players.

The PlayerData0 and PlayerData1 arrays are assumed (for code size reasons) to each exist within a page. MAX_PLAYERS*PLAYER_DATA_SIZE cannot exceed 256.

Redeye can only add/drop players during logon. If your game needs to have places where players can add in or drop out, you will need to re-enter logon, and after logon sort out who is who since all player numbers will be likely to be different.

For a while, there was talk of a "joystick only" version of redeye. After much thought about the problem, it seemed that the VAR_SIZE_DATA option provides better flexibility, since joystick and button values need be sent only when they are different.

Redeye logon variables are overlaid in the same space as the regular communication variables, so the game program must know which state it is in when referencing variables.

Receipt of the local system's messages must be handshaked in exactly the same way as all of the other players' data (even though the data is already known).

Redeye provides synchronization to the game frame, but this is not enough synch for precisely timed events, such as two machines playing different voices of the same piece of music. Due to possible communication problems, the game frame when using redeye is not guaranteed to happen at any particular rate. In practice, however, no problems have been observed.

Resource Usage

Redeye uses the serial port (timer 4) and two more general purpose timers (timers 1 and 5) during logon. After logon, redeye shuts down all three timers' interrupts. During normal communication, if only one player is playing, the timers are not used. If more than one player, then the serial port (timer 4) and one more general purpose timer (timer 1) are used.

During recent assembly of the testredeye program, the redeye system code (including variables and logon code) used the following amount of memory:

```
2103 + MAX_PLAYERS*(4+2*PLAYER_DATA_SIZE) bytes
If VAR_SIZE_DATA, add 50+(2*MAX_PLAYERS) bytes
If MAX_PLAYERS>8, add 110 bytes
```

The logon code (everything in the file comlink_logon.src) takes 745 bytes (an additional 20 bytes if MAXPLAYERS>8, and an additional 50 if REDEYE_SLOWNESS is non-zero), which can be reclaimed after logon. In addition, the example "glue code" in redeye_glue.src takes about 380 bytes.

During communication, redeye interrupts happen very frequently. A byte takes 176 usec to send (slightly longer than 1 scan line). Each message, redeye sends 3 bytes of overhead (1 byte each of size, header and checksum) and any data bytes the message may contain. If there is a problem (someone missed data, or someone is late with their data), extra messages will be sent. If frequent or long lasting interrupts other than redeye are running in the system, communication overhead can go way up. (The music and sound effects drivers have long interrupts, but both clear interrupt disable to minimize interference with redeye). If interrupt collisions are a problem, the REDEYE_SLOWNESS constant may be able to be adjusted to get better results.

Optimization

Shorter messages take less time to send and cause fewer interrupts. The use of VAR_SIZE_DATA to optimize message length can greatly improve system performance.

If code space is more important than system performance, some memory can be saved by not selecting VAR_SIZE_DATA .

Games using the glue code provided in the 6502:examples directory might have a noticeable lag in the responsiveness of the joystick if if they run a very long game frame. This has not been noticed an the current games. To reduce this potential lag, the switch SPLIT_SEND_RECEIVE can be defined. If SPLIT_SEND_RECEIVE is defined, then the routine preparejoysticks must be called in the game loop sometime after the routine do_comlink_joysticks . The goal is to set up the joystick data to be sent late in the game frame (preparejoysticks), but to still allow enough time for the Redeye communication to complete before attempting to read the data (do_comlink_joysticks).

If frequent or lengthy IRQs are happening, (playing digitized sounds or doing on-the-screen palette changes for example) Redeye communication can be degraded. To reduce the number of failed communication attempts, the constant REDEYE_SLOWNESS can be set to a number larger than 0 to slow down the rate at which redeye data is sent (and must be responded to). Each increment of REDEYE_SLOWNESS slows down communication by a factor of two. Slowing down the communication does not mean that the game slows down (all of the early Redeye games could have run with a much slower baud rate without noticing). With the REDEYE_SLOWNESS set to maximum (3), communication runs 8 times slower than normal, but still should provide enough bandwidth for a 4 player game sending up to two bytes of data (joystick & switches for example) to run the game loop at 15 frames/second. Setting REDEYE_SLOWNESS to a non-zero value adds 50 bytes to the logon code.

*NOTE: Setting the REDEYE_SLOWNESS switch to a non-zero value does not change the baud rate during logon, so Redeye's interrupt service requirements will remain high. Effects that require heavy interrupt usage (such as digitized sounds) may interfere with Redeye's logon code.

Redeye code grows significantly when supporting more than 8 players. The advantages of a 9 or 10 player game may not justify the code

ncrease.

Redeye code can shrink if some of its variables are moved to zero page. The routine ZeroComlinkVariables in comlink.src would need to be modified to reflect the change. Let me know if someone needs or is making this optimization.

The example program testredeye.src sets up OutGoingData only after the receipt of the message from the highest numbered player. This is fine when all of the redeye data is dealt with at once, and not distributed through the game logic. An optimization could be made by setting up OutGoingData and calling launch_redeye immediately after acknowledging receipt of the local system's message. NOTE: The value of Seq changes on every call to launch_redeye , but the messages not read from higher numbered players will still belong to the same sequence. This optimization is most effective when the game frame is long, redeye data is processed slowly throughout the game frame and the amount of data being sent is large.

In final release code, make sure that BRK_USER is not defined (unless your code uses the BRK instruction itself). Setting BRK_USER enables debugging capabilities on the Mandy/Pinky systems, but adds code and time overhead to the system IRQ handler.

Debugging

Multi-system code presents special difficulties in debugging, and is prone to a peculiar class of bugs not found in single-system code. In games that only pass joystick information, it is critical that every system agrees on the state of everything important to the game.

A technique found to be very useful in debugging "Gauntlet the Third Encounter" was to checksum the important variable space, and send the checksum along with the joystick data. After receiving a frame of data, the checksums were compared, and if any were different the game broke to the monitor for debugging. Important variables or state flags can also be passed around and checked this way.

Counters and random number generators should be initialized to a known state immediately following logon. Care should be taken to make sure that counters and random numbers do not change based on local or system specific reasons.